Sketchy Drawings

Marc Nienhaus
University of Potsdam
Hasso Plattner Institute
marc.nienhaus@hpi.uni-potsdam.de

Jürgen Döllner
University of Potsdam
Hasso Plattner Institute
juergen.doellner@hpi.uni-potsdam.de

Abstract

In non-photorealistic rendering sketchiness is essential to communicate visual ideas and can be used to illustrate drafts and concepts in, for instance, architecture and product design.
In this paper, we present a hardware-accelerated real-time rendering algorithm for drawings that sketches visually important edges as well as inner color patches of arbitrary 3D objects even beyond the geometrical boundary. The algorithm preserves edges and color patches as intermediate rendering results using textures. To achieve sketchiness it applies uncertainty values in image-space to perturb texture coordinates when accessing intermediate rendering results. The algorithm adjusts depth information derived from 3D objects to ensure visibility when composing sketchy drawings with arbitrary 3D scene contents. Rendering correct depth values while sketching edges and colors beyond the boundary of 3D objects is achieved by depth sprite rendering. Moreover, we maintain frame-to-frame coherence because consecutive uncertainty values have been determined by a Perlin noise function, so that they are correlated in image-space.
Finally, we introduce a solution to control and predetermine sketchiness by preserving geometrical properties of 3D objects in order to calculate associated uncertainty values. This method significantly reduces the inherent shower-door effect.

CR Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation – Display Algorithms I.3.3

Keywords: Non-photorealistic rendering, sketching, real-time rendering, image-space, hardware-acceleration, depth sprites.

1 Introduction

In non-photorealistic rendering (NPR), sketching is of vital importance to express the preliminary state of a draft, concept, or idea, especially in application areas such as architectural and product designs [Schumann et al. 1996].
Common photorealistic renditions are often less efficient in communicating ideas, outlines, and proposals, and they imply the impression of finality. These photorealistic renderings reduce one's ability to rethink enhancements and modifications.

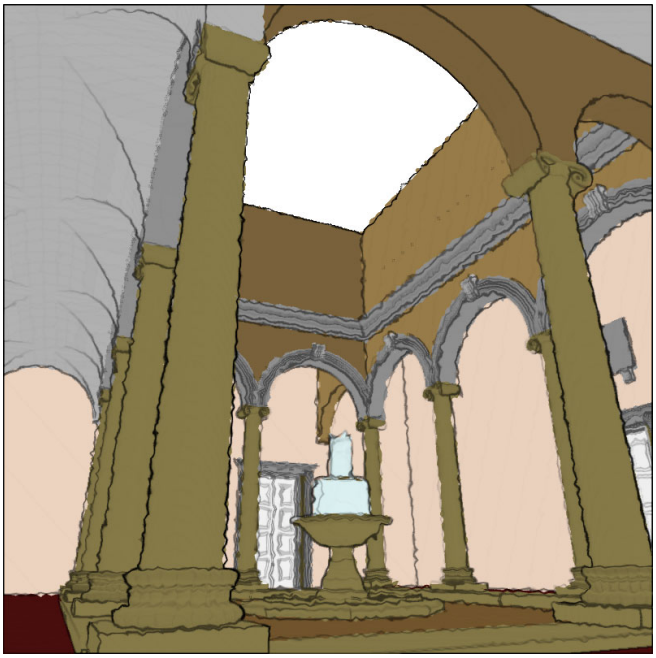

Figure 1: A sketchy drawing of a cloister generated with our real-time rendering algorithm.

In contrast, NPR communicates visually and is, therefore, more helpful when dealing with renditions. In particular, sketchy drawings encourage the exchange of ideas when people are reconsidering drafts and blueprints. Sketchy drawings express uncertainty and suggest work in progress. In fact, hand-drawn sketches are still an integral part of the development process in architectural or product design; in the film making process storyboards are still used to assist communication.

This paper presents a new general-purpose rendering algorithm to generate sketchy drawings of 3D scene geometry. Visually important edges and surface colors derived from the 3D object are sketched non-uniformly beyond the boundary of the original object. Generally speaking, our algorithm (1) sketches the outline of 3D objects to imply vagueness and (2) crayons in inner color patches exceeding the sketchy outline as though they have been painted roughly (Fig. 1, 11). The algorithm is used most importantly to express ideas using uncertainty. Moreover, it supports variations in style.

The algorithm represents an image-space algorithm and is designed to use the resources provided by modern graphics accelerators. Furthermore, the output generated with the algorithm is general with respect to real-time rendering using raster graphics. Thus, the algorithm

- can process arbitrary 3D geometry

- runs in real-time
- gives results that can be combined with general 3D scene contents

The remainder of this paper is structured as follows: Section 2 discusses related work. Section 3 describes the sketchy drawing algorithm in detail. Section 4 presents an approach to control uncertainty. Section 5 draws conclusions and discusses future work.

2 Related Work

NPR has become a popular research topic in computer graphics for the last decades. Non-photorealistic rendering styles include painterly rendering [Hertzmann 1998], hatching [Praun et al. 2001], and edge-enhancement [Nienhaus and Döllner 2003]. A number of non-photorealistic rendering algorithms focus on conveying and illustrating 3D geometry [Gooch et al. 1998; Gooch et al. 1999] while others concentrate on sketching [Curtis 1998; Haddon 2002].

An increasing number of conceptually new non-photorealistic rendering algorithms are being designed to use the resources and computational frequencies available in the rendering pipeline and accessible via today's graphics hardware. They achieve real-time performance. Examples include the work of Freudenberg et al. [2002], Praun et al. [2001], and Raskar [2001]. Our algorithm represents one of the first algorithms of this category addressing sketchy drawings of arbitrary 3D geometry. It takes advantage of Higher-Level Shading Languages, but can not be implemented using shaders only.

Decaudin [1996] introduces an image-space algorithm for detecting edges of 3D geometry to achieve cartoon-style depictions. His algorithm is based on the G-buffer concept introduced by Saito and Takahashi [1990]. G-buffers are 2-dimensional data structures that store geometrical properties of 3D geometry. Important G-buffers are the normal buffer, the z-buffer, and the Id-buffer. Image processing operations are provided with G-buffers to analyze their contents and produce comprehensible images of 3D objects. Decaudin utilizes image-processing techniques to extract discontinuities in the normal and z-buffer that from the edges of 3D scene geometry. Since image processing techniques are time consuming, his approach is not processed in real-time.

Image-based rendering has been accelerated, now being usable for real-time image processing operations even for non-photorealistic rendering [Mitchell 2003]. Mitchell et al. [2002] present a hardware-accelerated real-time image processing technique for extracting edges and enhancing images on a per-scene basis. Their technique renders fragment normals, z-values, and object identifiers of 3D geometry into textures using a render-to-texture implementation. It then detects discontinuities in these buffers using graphics hardware and combines the resulting edges with framebuffer contents. Edges of 3D scene geometry, regions in shadow, and texture boundaries can be outlined using their method.

Our sketchy drawing algorithm is based on our edge-enhancement algorithm [Nienhaus and Döllner 2003] that also takes advantage of hardware-acceleration. The edge-enhancement algorithm determines edges on a per-object basis. It distinguishes between profile edges and edges of inner forms by handling discontinuities in the normal and z-buffer differently. The assembly of intensity values constitutes edges that are rendered into a texture, called edge map. The algorithm preserves the edge map, so that it can be combined with manifold non-photorealistic rendering algorithms [Praun et al. 2001; Gooch et al. 1998; Freudenberg et al. 2002] and advanced multipass, real-time rendering algorithms [Blythe et al. 1999]. The present work constructs the edge map in an intermediate rendering pass and uses it as one ingredient for generating a sketchy outline. In general, preserving intermediate rendering results as textures on a per-object basis allows us to restrict the effect to individual objects and to compose the effect in a subsequent rendering pass of our multipass rendering algorithm.

Strothotte et al. [1990] argue in favor of uncertainty when visualizing ancient architecture. They implement a system for visualizing and outlining ancient architecture using a sketch renderer. Our work implements uncertainty to suggest the preliminary state of a draft or idea. In order to do this, we determine uncertainty values in image-space based on the Perlin noise function [Ebert et al. 1998; Perlin 1985]. These are applied to perturb intermediate rendering results in image-space. So our sketchy drawings express "imprecision, incompleteness, and vagueness" when rendering architecture or technical objects. By means of correlated Perlin noise values our algorithm maintains frame-to-frame coherence.

Curtis [1998] presents a *Loose and Sketchy* filter that sketches the edges of 3D geometry using image processing. The filter uses a depth map as input and converts it into a template image and into a force field image. The template image determines the amount of ink needed in the neighborhood of a pixel, whereas the force field image affects the movement of particles along edges. To generate sketches of various styles, particles are placed randomly in image-space that move along edges, adding or erasing ink until they die. Curtis' loose and sketchy filter is not meant to run in real-time.

Northrup and Markosian [2000] introduce a real-time silhouette-rendering algorithm using Artistic Strokes. Their hybrid algorithm determines potential silhouette edges and computes their visibility by sampling an ID reference image. Visible edges are then inserted into the 3D scene as triangle strips using stylistic variations. In this method the inserted geometry aligns loosely to the original geometry. Artistic strokes are thus rendered beyond 3D geometry resulting in stylistic depictions. Artistic Strokes have been elaborated by Kalnins et al. [2002]. They implemented a WYSIWYG system that allows designers to annotate 3D objects using brush strokes to generate aesthetic non-photorealistic renderings. They also observed that the number of strokes influences performance because of sampling the ID reference image. Furthermore, Kalnins et al. [2003] provide a solution for temporal coherence of stylistic silhouettes for objects of moderate complexity.

Our algorithm also sketches edges beyond the boundary of 3D scene geometry. The edges align loosely to the object. In fact, our algorithm is one of the first image-space real-time rendering algorithms that allows for stylized edges [Isenberg et al. 2003]. Furthermore, the sketches we propose are not limited to edges. They also include inner color patches, which are derived from surface colors provided with 3D geometry. Thus, color patches are rendered beyond sketched edges that outline inner forms and the boundary of the object. Sketching both edges and color patches have rarely been addressed by previous work. Furthermore, our image-space algorithm is, by its nature, both almost independent of the complexity of the object, and totally independent of the number of edges that are sketched. In addition, it needs very few prerequisites from 3D geometry. Finally, our algorithm both maintains frame-to-frame coherence and provides a way of reducing the shower-door effect.

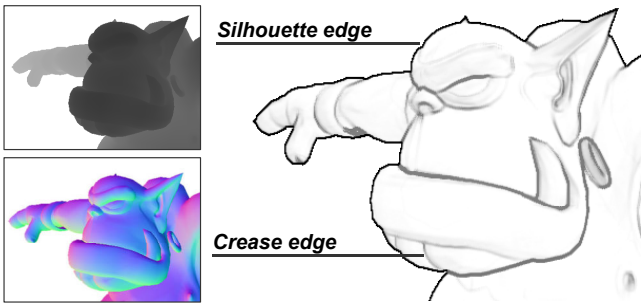

Figure 2: Sampling neighboring texels in textures representing the normal-buffer and the z-buffer (left) allows for extracting discontinuities resulting in edge intensities stored in the edge map (right).

By means of depth sprites, our algorithm generates and adjusts depth information. Hence, we can implement sketchiness by rendering those parts of the 3D object that exceed its original boundary. Depth sprites enable us to combine sketchy drawings arbitrarily with further 3D scene geometry.

Due to its implementation and its per-object basis, sketchy drawings can easily be integrated into any real-time graphics applications such as CAD or storyboarding systems.

3 Sketchy Drawing

Our sketchy drawing algorithm considers visually important edges and surface colors to sketch 3D scene geometry. Both are sketched non-uniformly using uncertainty.

The algorithm proceeds as follows. (1) It generates intermediate rendering results that represent the edges and surface colors of 3D geometry. (2) It applies uncertainty values in image-space to sketch intermediate rendering results non-uniformly. (3) It adjusts depth information so that the resulting sketchy drawing can be merged with general 3D scene contents.

3.1 Intermediate Rendering Results

We denote 2-dimensional data derived from 3D geometry and rendered into textures as intermediate rendering results; they are reused in subsequent rendering passes.

As ingredients for sketchy drawings we primarily consider (1) visually important edges and (2) surface colors, both provided as intermediate rendering results.

Visually important edges include silhouette, border, and crease edges. We obtain these edges by extracting discontinuities in the normal buffer and z-buffer (Fig. 2). To achieve this, encoded normal and z-values of 3D geometry are rendered directly into 2D textures. So, as a prerequisite, 3D geometry must provide per-vertex normals. We then texture a screen-aligned quad that fits completely into the viewport of the canvas using the preceding textures. We calculate texture coordinates (s,t) of each fragment produced for the quad in such a way that they correspond to windows coordinates. Sampling neighboring texels allows us to extract discontinuities that result in intensity values that constitute the edges of 3D geometry. We render the assembly of edges into a single texture, that we call *edge map*. Figure 2 depicts the normal and z-buffer and the resulting edge map.

We render unlit 3D geometry while taking into account its color. This results in striking color patches that appear flat, cover all surface details, and emulate a cartoon-like style. We render the color of 3D geometry directly into a texture. This texture represents inner color patches of that geometry. We refer to that texture as *shade map* (Fig. 3).

3.2 Sketching using Uncertainty Values

Sketchiness is managed by uncertainty values applied to the edges and surface colors. We once again texture a screen-aligned quad using edge and shade maps as textures. To simulate the effect of "sketching on a flat surface" we apply uncertainty values in image-space to perturb texture coordinates of each fragment of that quad.

We thus apply an additional texture, whose texture values represent uncertainty values. Since we want to achieve frame-to-frame coherence, we opt for a noise texture whose texture values have been determined by a Perlin function; thus neighboring uncertainty values are correlated in image-space. Once created in a preprocessing step, the noise texture serves as an offset texture for accessing the edge and shade maps when rendering, i.e., its texture values slightly perturb texture coordinates that access the edge and shade maps.

Furthermore, we introduce a degree of uncertainty in order to control the amount of offset when accessing the edge and shade maps. To texture the quad, we multiply uncertainty values derived from the noise texture by a predefined 2×2 matrix used to weight these values. The result is an offset vector that translates texture coordinates. Figure 4 illustrates the perturbation of the texture coordinates accessing the shade map using the degree of uncertainty.

To emphasize sketchiness, we perturb texture coordinates for accessing the edge map and shade map differently. Thus, we apply two different 2×2 matrices and this results in different degrees of uncertainty. One degree of uncertainty shifts texture coordinates of the edge map, and one shifts texture coordinates of the shade map. Figure 3 illustrates the edge and shade maps after uncertainty has been applied.

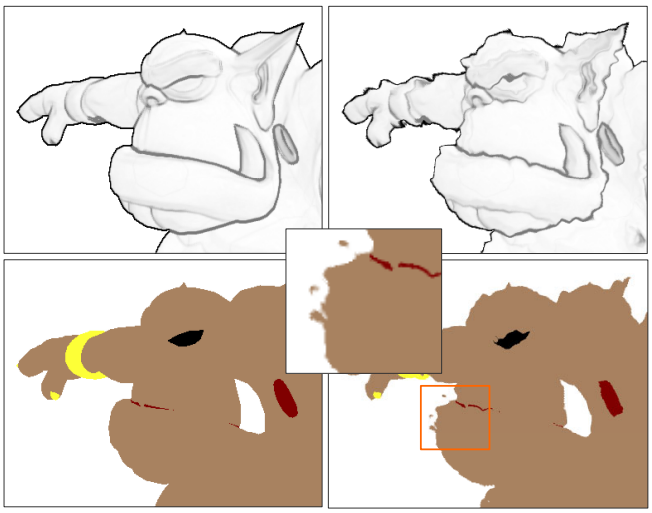

Figure 3: The edge map (upper left) and shade map (lower left) are two ingredients for sketchy drawings. Applying uncertainty results in perturbations of the edge map (upper right) and perturbations of the shade map (lower right). A magnification of the perturbed shade map illustrates spots produced beyond the boundary of 3D scene geometry.

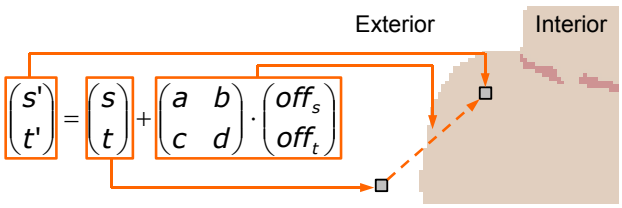

Figure 4: The uncertainty value (off_s,off_t) derived from the noise texture multiplied with a 2×2 matrix (with weights a, b, c and d) forms the degree of uncertainty that is applied to the texture coordinates (s,t) of a fragment to shift texture coordinates in image space. In this case, the perturbed texture coordinates (s',t') access a texture value of the interior region of the shade map even though the initial texture coordinates (s,t) would access the exterior region.

We classify texture values representing fragments of 3D geometry as *interior regions*, and texture values that do not correspond to fragments of 3D scene geometry as *exterior regions*.

Texturing a screen-aligned quad and perturbing the texture coordinates using uncertainty values allows us to access the interior regions of the edge and shade maps, whereas the initial texture coordinates would access exterior regions and vice versa (Fig. 4). The interior regions can thus be sketched beyond the boundary of 3D scene geometry, and exterior regions can penetrate interior regions. We can even produce spots beyond the boundary of the 3D geometry. This effect can be observed in the magnification in Figure 3.

Finally, we combine texture values of both the edge and the shade map. Multiplying the intensity values derived from perturbing the edge map with the color values derived from perturbing the shade map forms the basis for our sketchy drawing.

The uncertainty values (off_s,off_t) generated for the sketchy drawing in Figure 5 are calculated by the turbulence function, which is based on a Perlin noise function:

```
off_s ← turbulence(s,t);
off_t ← turbulence(1-s, 1-t);
```

3.3 Adjusting Depth Information

So far we have generated sketchy drawings by texturing a screen-aligned quad. This approach has significant shortcomings.

When rendering a screen-aligned quad textured with the texture of 3D geometry, (1) z-values of the original geometry are not available. Moreover, (2) the depth information of the original geometry is not available in the exterior regions when uncertainty has been applied.

To overcome these shortcomings, we adjust z-values using depth sprites. Conceptually, depth sprites are 2-dimensional images that provide an additional z-value at each pixel. To facilitate depth sprite rendering we implement a specialized fragment shader [Kilgard 2003; Rost 2004].

In general, depth-sprite rendering works as follows:

1) We capture z-values of the 3D geometry into a high precision depth texture, called *depth map* (Fig. 6).

2) We render a screen-aligned quad using the depth map as its texture. In this way we replace fragment z-values (produced by the rasterizer) with depth map values using the fragment shader.

For sketchy drawings, we have to modify the fragment shader to allow for the previous perturbations. So we access the high precision depth map twice and perturb texture coordinates of the quad. As first perturbation, we take the degree of uncertainty used for accessing the edge map; as second perturbation we take the degree of uncertainty used for accessing the shade map. The minimum value of both these texture values is used as the final fragment z-value for depth testing.

Figure 6 illustrates the combination of both perturbations applied to the depth map. The interior region of the perturbed depth map matches the combination of the interior regions of both the perturbed edge map and the perturbed shade map. Even those spots produced by perturbing the shade map (Fig. 3) appear in the perturbation of the depth map, as can be observed in the magnification.

The fragment shader calculates the modified z-values to render the textured quad that represents the sketchy drawing using an ordinary depth test. This way, the z-buffer remains in a correct state, and sketchy drawings can be arbitrarily composed with further (e.g., non-sketchy) 3D geometry. The accompanying video illustrates this feature.

3.4 Variation in Style

In the following, we present two variations in style that demonstrate the versatility of our algorithm. Minor changes to the original algorithm allow us to vary the style of sketchy drawings.

Sketching Edges Repeatedly. A fundamental technique in hand drawings is to repeatedly draw the edges in order to emphasizes

Figure 5: Applying different degrees of uncertainty perturb the edge and shade map non-uniformly in image-space. Combining the results forms the final sketchy drawing. Depth sprites allow sketchy drawings to be composed with a 3D scene.

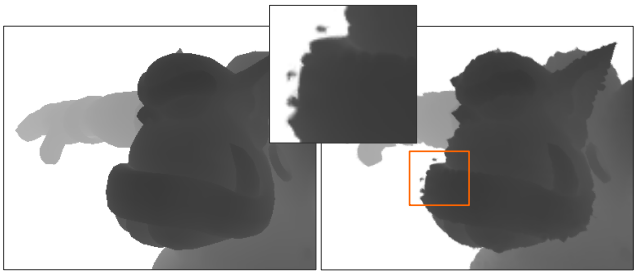

Figure 6: Depth sprites utilize the depth map (left) to adjust z-values. To utilize depth sprites for sketchy drawings we reproduce the degrees of uncertainty applied to the edge and shade map to perturb the depth map (right). The profile in the magnification illustrates spots resulting from the degree of uncertainty applied the shade map.

the preliminary state of a draft [Cabarga 1993].

We can simulate this technique by sketching visually important edges only. For that purpose, we exclude the shade map but apply the edge map multiple times using different degrees of uncertainty and possibly different edge colors. Edges thus overlap non-uniformly as if the edges of 3D geometry have been sketched repeatedly. Clearly, depth information must be adjusted by accessing the depth map multiple times, using the correspondent degrees of uncertainty. Figure 7 shows a sketchy drawing with two repetitions.

Roughening Profiles and Color Transitions. Although visually important edges and surface colors are sketched non-uniformly, the profiles and the color transitions of a sketchy drawing are exactly as if sketched with ink-pencils on a flat surface. We roughen the profiles and color transitions to simulate different drawing tools and media, for instance, chalk applied on a rough surface.

We apply a noise texture whose consecutive texture values are uncorrelated:

$$off_s \leftarrow \texttt{random();}$$
$$off_t \leftarrow \texttt{random();}$$

Thus, the degree of uncertainty applied to the texture coordinates of consecutive fragments that access the edge and shade maps are also uncorrelated.

This approach results in a sketchy drawing with softened and frayed edges and color transitions as illustrated in Figure 8. It can be observed that the roughness and granularity – especially for the darker edges – varies as though the pressure had varied as it does when drawing with chalk. This effect depends on the amount of uncertainty applied in image-space. The sketchy drawing shown in Figure 8 is still real-time capable.

3.5 Implementation Details and Performance

The rendering algorithm we have presented requires, in general, 4 passes: 2 rendering passes to render 3D objects into textures (1); 1 intermediate rendering pass to render a screen-aligned quad to extract edge intensities (2); 1 final pass to render a screen-aligned quad as a depth sprite for sketching in image-space (3).

In 1), 2), we capture the contents of a non-visible frame-buffer into textures. To do so, a pbuffer can serve as a render-to-texture implementation [Kilgard 2003].

1) In the first rendering pass our algorithm renders 3D geometries with encoded per-fragment normals and z-values. We capture the normal buffer in one texture and the z-buffer in a high precision depth texture. In the second rendering pass, we render the color values of the 3D geometries to generate the shade map.

2) In the intermediate rendering pass the algorithm renders a screen-aligned quad textured with the normal and depth texture to extract edge intensities. The result directly represents the edge map.

3) Finally, a screen-aligned quad is rendered as depth sprite into the visible framebuffer to compose the sketchy drawing. Since the high precision depth map has already been generated in the first rendering pass, we can reuse it for sketching.

In an optimized version, our algorithm uses float-buffers to merge the first and the second rendering pass. A float-buffer provides 32 bits precision for each RGBA channel, and its contents can be reused as a float-texture with the very same precision [Kilgard 2003]. Fragment programming allows for packing and unpacking ordinary 8 bit RGBA color values using just one channel of the float-buffer or float-texture. Our algorithm packs encoded per-fragment normals into the R-channel and the color values of the 3D objects into the G-channel, and finally, it directs the uncompressed z-value into the B-channel of a 32-bit RGB float-texture. Thus, one single texture contains the normal buffer, the shade map, and the high precision depth map for further use.

We accelerate our algorithm slightly by rejecting unneeded fragments produced by the rasterizer when rendering a screen-aligned quad. If a fragment's z-value derived from accessing the depth map equals 1 – which denotes the depth of the back clipping plane of the view frustum – the fragment shader rejects that fragment in advance, so that we can optimize the fill rate.

Overall, the algorithm can take advantage of hardware-acceleration such as render-to-texture, multi-texturing, fragment shading, float-buffers, and float-textures. Hence we achieve real-

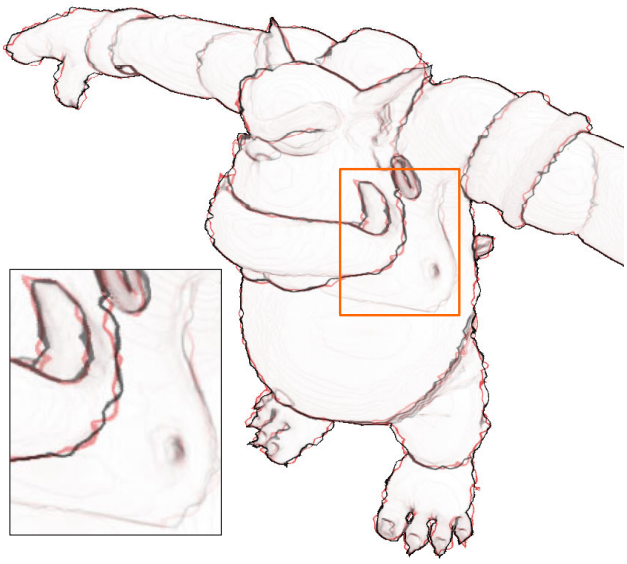

Figure 7: A sketchy drawing generated by sketching visually important edges repeatedly. The color value for each edge can be chosen individually.

Figure 8: Sketchy drawing simulating chalk applied on a rough surface. Variations in edges intensities let assume that the pressure applied when drawing them has varied.

time frame rates. The model of the "Ogre" in the preceding figures can be rendered at 20 fps at a window resolution of 800×800 using an NVIDIA GeForce FX 5600 graphics card. It should be noted that this performance is almost independent of the CPU.

4 Controlling Uncertainty

Controlling uncertainty values, in general, enables one to configure and design the visual appearance of sketchy drawings.

In the previous chapter, we showed how we provide uncertainty values based on a Perlin noise function for each pixel in image-space. This has the following benefits:

- we achieve frame-to-frame coherence, for instance, when interacting with the scene, since neighboring uncertainty values are correlated
- we can access the interior region from beyond the exterior region and vice versa. It thus allows sketching beyond the boundary of the 3D object

However uncertainty values remain unchanged in image-space and have no obvious correspondence with geometrical properties of the targeted 3D geometry. So sketchy drawings tend to "swim" in image-space (shower-door effect) and their visual appearance cannot be predetermined.

To overcome these limitations, we have to accomplish both:

- preserve geometrical properties such as surface positions, normals, or curvature information to determine uncertainty values
- continue to provide uncertainty values in the exterior region, at least close the 3D geometry

4.1 Preserving Geometrical Properties

The principle outline of our approach to the preservation of geometrical properties to control uncertainty is as follows:

1) we render geometrical properties directly into a texture that forms an additional G-buffer
2) we texture a screen-aligned quad with that texture, and access geometrical properties via texture coordinates (s,t)
3) we calculate uncertainty values based on – to simplify matters – a noise function using geometrical properties as parameters

These uncertainty values can then be used to determine different degrees of uncertainty to generate perturbations that produce texture coordinates (s',t'). Mathematically, the algorithm determines the perturbed texture coordinates (s',t') by the following function:

$$f:(s,t)\rightarrow(s',t')$$
$$f(s,t)=p(s,t,g(s,t))$$

where (s,t) are the texture coordinates of a fragment produced when rasterizing the screen-aligned quad, $g(\cdot)$ corresponds to geometrical properties available in the additional texture, and $p(\cdot)$ determines the perturbation applied to (s,t) using $g(\cdot)$ as input.

Note that there are two functions $f(s,t)$ to handle perturbations to access the edge ($f_{Edge}(s,t)$) and the shade ($f_{Shade}(s,t)$) maps separately.

4.2 Expanding the Objects Boundary

We enlarge the original sized 3D geometry to generate geometrical properties in the surrounding of the object. To do this we slightly shift each vertex of the 3D geometry along its vertex normal in object-space.

Figure 9: A sketchy drawing generated by using uncertainty values that are based on geometrical properties of 3D scene geometry.

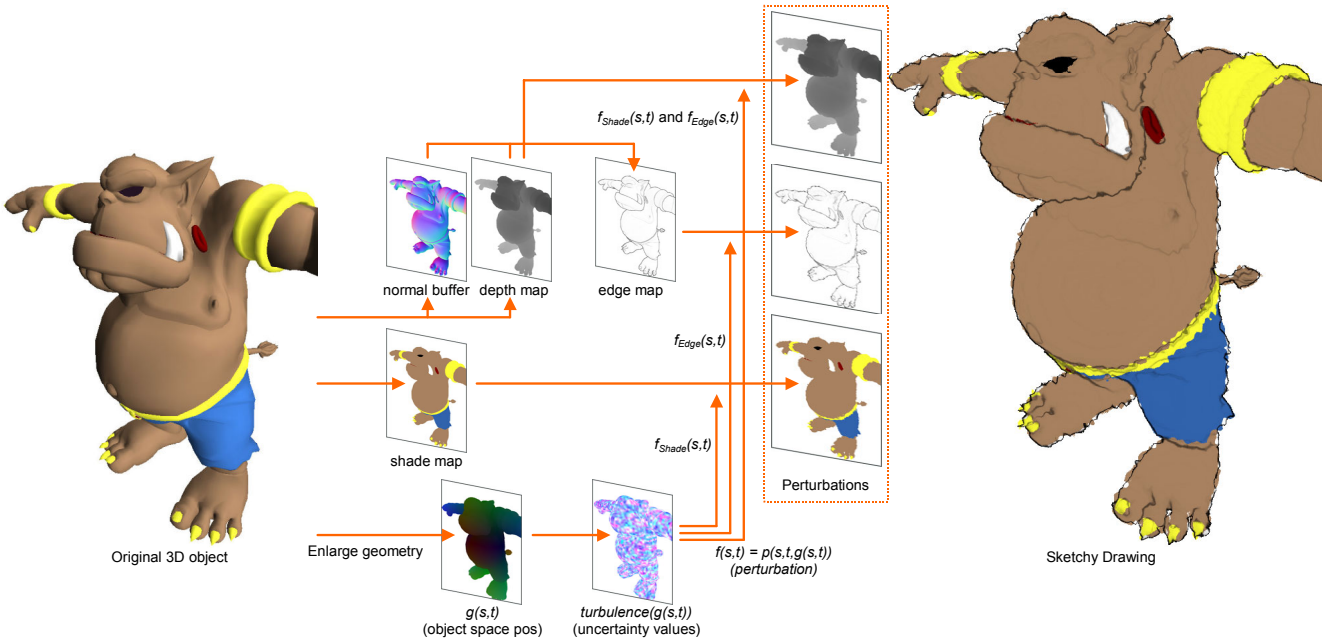

Figure 10: The conceptual sketch illustrates buffers and intermediate rendering results involved in the process of generating sketchy drawings. Furthermore, it clarifies the usage of f(s,t) when considering geometrical properties.

For this to be possible, the 3D geometry must satisfy the following requirements:

- the surface forms a connected component
- each shared vertex provides an interpolated normal

Enlarging the 3D geometry allows us to render geometrical properties into a texture to calculate uncertainty values in the interior regions as well as in the exterior regions. Again, the interior regions can be sketched beyond the boundary of the 3D geometry and the exterior regions can penetrate the interior regions. But this time the perturbations are based on uncertainty values that do have an obvious correspondence to the underlying 3D geometry.

4.3 Managing the Shower-Door Effect

The following example demonstrates how sketchiness can be controlled to reduce the shower-door effect.

Enlarged 3D geometry is rendered with object-space positions as color values into a texture. We achieve this by determining the object-space position for each displaced vertex and providing them as texture coordinates to the rasterization process. The rasterizer produces interpolated object-space positions for each fragment using barycentric coordinates. A specialized fragment shader directs them as high precision color values to a targeted float-buffer. Thus, $g(s,t)$ preserves objects space positions as a float-texture.

Based on $g(s,t)$, the algorithm can determine texture coordinates $f(s,t)$ using $p(\cdot)$. In our example, the function $p(\cdot)$ processes the perturbation using a predefined 2×2 matrix and a Perlin noise function encoded into a 3D texture. Thus, $p(\cdot)$ accesses the 3D texture using $g(s,t)$ as texture coordinates. The resulting texture value is multiplied by the 2×2 matrix to obtain a degree of uncertainty. $f(s,t)$ applies the degree of uncertainty to perturb (s,t) resulting in (s',t').

Calculating $f_{Edge}(s,t)$ and $f_{Shade}(s,t)$ using different matrices results in the sketchy drawing (Fig. 9). The accompanying video illustrates that the shower-door effect has been significantly reduced.

In the first case, the example aims at reducing the shower-door effect. But, in the second case, it gives a clue as to how to control the appearance of sketchy drawings using geometrical properties. By means of higher-level shading languages provided by OpenGL 2.0 [Rost 2004] or Cg [Mark 2003] one could design sketchy drawings more intuitively.

The diagram in Figure 10 illustrates the process flow of our algorithm for generating sketchy drawings by considering geometrical properties.

5 Conclusion

We have presented a real-time rendering algorithm for generating sketchy drawings of arbitrary 3D geometries. The results depict imprecision, incompleteness, and vagueness to communicate drafts, concepts and ideas.

Only minor changes to the general algorithm are needed in order to produce variations in the visual appearance of sketchy drawings. Examples include sketching edges repeatedly as well as roughening profiles and color transitions.

We introduced perturbations applied to depth sprites to ensure a correct depth behavior – particularly in the exterior regions. This allows us to combine sketchy drawings with any kind of real-time rendering application. Generally, operations applied to depth textures for depth sprite rendering offer manifold applications when combining image-based effects and arbitrary 3D geometry. Thus the contribution of our algorithm is not limited to NPR.

Technically, the algorithm represents a hardware-accelerated algorithm designed for today's graphics cards. Our algorithm produces and accesses geometrical properties as well as uncertainty values in the exterior regions using intermediate rendering results to produce sketchy drawing effects. Therefore, shaders written in Higher-Level Shading Languages cannot substitute our sketchy drawings algorithm.

In our future work, we expect to produce sketchy drawings that appear more artistically pleasing using texture lookups into brush strokes. Furthermore, we aim at designing sketchy drawings procedurally using Higher-Level Shading Languages to target different applications areas such as storyboard depictions.

6 References

BLYTHE, D., GRANTHAM, B., KILGARD, M. J., MCREYNOLDS, T., AND NELSON, S. R. 1999. Advanced Graphics Programming Techniques Using OpenGL. In *ACM SIGGRAPH 1999 Course Notes*.

CABARGA. L. 1993. Dynamic Black & White Illustration – One Hundred Years of Line Art 1900 - 2000. *Art Direction Books*, New York.

CURTIS, C. 1998. Loose and Sketchy Animation. In *ACM SIGGRAPH 1998 Conference Abstracts and Applications*, p. 317.

DECAUDIN, P. 1996. Rendu de scénes 3D imitant le style «dessin animé». *Rapport de Recherche 2919*. Université de Technologie de Compiègne, France.

EBERT, D. S., MUSGRAVE, F. K., PEACHEY, D., PERLIN, K., AND WORLEY, S. 1998. *Texturing & Modeling – A Procedural Approach (Second Edition)*, Academic Press Professional, Inc., San Diego, CA.

FREUDENBERG, B., MASUCH, M., AND STROTHOTTE, T. 2002. Real-Time Halftoning: A Primitive For Non-Photorealistic Shading. *13th Eurographics Workshop on Rendering*. Pisa, Italy, pp. 1-4.

GOOCH, A., GOOCH, B., SHIRLY, P., AND COHEN, E. 1998. A Non-Photorealistic Lighting Model for Automatic Technical Illustration. In *Proceedings of ACM SIGGRAPH 1998*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 447-452.

GOOCH, B., SLOAN, P. S., GOOCH, A., SHIRLEY, P., AND RIESENFELD, R. 1999. Interactive Technical Illustration. *ACM Symposium on Interactive 3D Graphics 1999*, pp. 31-38.

HADDON, J. 2002. Sketchy Rendering. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, New York.

HERTZMANN, A. 1998. Painterly Rendering with Curved Brush Strokes of Multiple Sizes. In *Proceedings of ACM SIGGRAPH 1998*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 453-460.

ISENBERG, T., FREUDENBERG, B., HALPER, N., SCHLECHTWEG, S., AND STROTHOTTE, T. 2003. A Developers's Guide to Silhouette Algorithms for Polygonal Models. *IEEE Computer Graphics and Applications*, 23(4), 28-37.

KALNINS, R. D., MARKOSIAN, L., MEIER, B. J., KOWALSKI, M. A., LEE, J. C., DAVIDSON, P. L., WEBB, M., HUGHES, J. F., AND FINKELSTEIN, A. 2002. WYSIWYG NPR: Drawing Strokes Directly on 3D Models. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2002)*, 21(3), 755-762.

KALNINS, R. D. DAVIDSON, P. L., MARKOSIAN, L., AND FINKELSTEIN, A. 2003: Coherent Stylized Silhouette. *ACM Transactions on Graphics*, 22(3), 856-861.

KILGARD, M. (Ed.). 2003: *NVIDIA OpenGL Extension Specifications*. NVIDIA Corporation, June 2003. http://deveoper.nvidia.com/docs/IO/1174/ATT/nvOpenGLspecs.pdf

MARK, W. R., GLANVILLE, R. S, AKELEY, K., AND KILGARD, M. J. 2003: Cg: A System for Programming Graphics Hardware in a C-like Language. *ACM Transactions on Graphics (Proceedings of ACM SIGGRAPH 2003)*, 22(3), 896-907.

Figure 11: Several sketchy drawings of a cloister captured from different viewpoints.

MITCHELL, J. L. 2003: Real-Time 3D Scene Post-processing. Game Developers Conference, San Diego, CA. www.ati.com/developer/gdc/GDC2003_ScenePostprocessing.pdf

MITCHELL, J. L., BRENNAN, C., AND CARD, D. 2002: Real-Time Image Space Outlining for Non-Photorealistic Rendering. In *ACM SIGGRAPH 2002 Conference Abstracts and Applications*, 239.

NIENHAUS, M. AND DÖLLNER, J. 2003. Edge-Enhancement – An Algorithm for Real-Time Non-Photorealistic Rendering. *Journal of WSCG'03*, 346-353.

NORTHRUP, J. D. AND MARKOSIAN. L. 2000. Artistic Silhouettes: A Hybrid Approach, In *Proceedings of the First International Symposium on Non-Photorealistic Animation and Rendering (NPAR 2000)*, 31-38.

PERLIN, K. 1985. An image synthesizer, In *Computer Graphics, 19(3), (Proceedings of ACM SIGGRAPH 1985)*, ACM, 287-296.

PRAUN, E., HOPPE, H., WEBB, M., AND FINKELSTEIN, A. 2001: Real-Time Hatching. *Proceedings of ACM SIGGRAPH 2001*, ACM Press / ACM SIGGRAPH, New York, Computer Graphics Proceedings, Annual Conference Series, ACM, 579-584.

RASKAR, R. 2001. Hardware Support for Non-photorealistic Rendering. In *Proceedings of ACM SIGGRAPH/Eurographics Workshop on Graphics Hardware (2001)*, 41-46.

ROST, R. J. 2004: *OpenGL® Shading Language*. Addison-Wesley Professional.

SAITO, T. AND TAKAHASHI, T. 1990: Comprehensible Rendering of 3-D Shapes. In *Computer Graphics (Proceedings of ACM SIGGRAPH 1990)*, 24(4), 197-206.

SCHUMANN, J., STROTHOTTE, T., RAAB, A., AND LASER, S. 1996. Assessing the Effect of Non-photorealistic Rendered Images in CAD. In *Proceedings of SIGCHI 1996 Conference on Human Factors in Computing Systems*, 35-41.

STROTHOTTE, T., MASUCH, M., AND ISENBERG, T. 1990: Visualizing Knowledge about Virtual Re-constructions of Ancient Architecture. *Proceedings of CGI 1999*, 36-43.